YOLO Distillation for Real-Time Inference on a Raspberry Pi Zero

Theo Coulson
Northwestern University
Evanston, USA
theocoulson@u.northwestern.edu

Rishika Bera
Northwestern University
Evanston, USA
rishikabera2026@u.northwestern.edu

Abstract

We used a fine-tuned YOLO-11n as the teacher model for a custom model architecture for detection and bounding box estimation. Our architecture was designed to be sufficiently lightweight to run in real time on a raspberry pi zero alongside the drone's main control code, to allow it to automatically detect and land on a landing platform. We achieved reasonable detection performance with one-tenth of the teacher model's parameters.

Our code can be found here: https://github.com/theoHC/yolo-berry-distillery

Keywords: YOLO, Optitrack, Small Drone, Distillation

ACM Reference Format:
Theo Coulson and Rishika Bera. 2026. YOLO Distillation for Real-Time Inference on a Raspberry Pi Zero. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Detection and bounding-box estimation models like YOLO make a number of common computer vision tasks extremely accessible, thanks to the easy-to-use fine tuning framework provided by Ultralytics, and the strength of the model backbone, which enables high performance on custom tasks with very small training datasets. However, most YOLO models require some sort of graphics card to run at a reasonable speed, and take up gigabytes of ram. While acceptable for fast inference on most modern laptops, all but the very smallest YOLO models are completely out of reach for embedded systems like the raspberry py zero, whose most sophisticated incarnation, the 2w, has only half a gygabyte of ram and 4 1 gigahertz cpu cores. Our project demonstrates that simplifying task-specific assumptions allowed us to slash model size while retaining acceptable detection performance.

2 UAV Platform

Figure 1. Small Drone used in the project

The drone system we are using has been developed by the lab of Mike Rubenstein (principally Andrew Curtis) for work on robotic swarms. We have been given a drone integrated with this system which we are able to use for this poject. The system involves three components: a custom Optitrack interface which broadcasts position information over wi-fi, a ground station for managing the fleet, and the drone itself.

The drone is principally controlled using a set of Python scripts which interface with the Optitrack position signals,

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

other drones, the ground station, and the flight controller. These scripts provide an API for user code, which is able to access information about the drone and its environment from the optitrack and send commands to the flight controller in the form of position setpoints and velocity constraints.

For this project, a new drone was built according to the standard pattern for its type, with the addition of the raspberry pi camera module secured to the underside by double-sided tape. Unfortunately, we were not able to integrate the model into the drone control system as our vehicle encountered a number of issues with the serial connection to the flight controller, which we were not able to fix in time for this submission.

3 Downscoping from our Initial Pitch

When we first pitched this project, we aimed to get a miniaturized VLA running on the drone, which would involve distilling it and, most likely, adapting it to better integrate with Optitrack data. We downscoped to the much more technically limited project for a few reasons. Firstly, further analysis made clear that running on the pi would require so much shrinkage that preserving functionality would be extremely unlikely. Secondly, the optitrack-based api for the drones simply didn't provide very many things for a VLA to do that would meaningfully reflect its capabilities. Thirdly, gathering and properly annotating training data for this platform would have been prohibitively time intensive. We also had a team member loose nearly two weeks to illness. We therefore pivoted towards YOLO distillation as a more achievable and testable aim.

4 Custom Architecture

YOLO11 is available with between 2.6 and 56 million parameters, depending on the version. The smallest, YOLO11n, almost completely fills the available memory of the Raspberry Pi Zero 2W and takes several seconds per inference. A traditional YOLO architecture has three parts: a backbone, a neck, and a head. The backbone extracts features at multiple spatial scales (P3, P4, P5). The neck fuses these scales together using a Feature Pyramid Network (FPN) so both fine spatial detail and high-level semantics are available at every scale. The head then produces the bounding box and class predictions. Our task only required detecting a single object, the landing pad, which has known and roughly fixed dimensions. Since the neck exists mainly to fuse features across scales so a model can handle objects of varying sizes, it serves no purpose for us. We eliminate it entirely and extract features at a single scale.

4.1 Student Model

The complete student model is built from the components described below. The backbone extracts features from the 224x224 input image, spatial pooling reduces the result to a 4x4 grid, and the head maps this to the confidence score and box coordinates. With no neck and no multi-scale grid, the model contains only a fraction of the parameters of even the smallest YOLO11 variant, allowing it to run in real time within the limits of the Raspberry Pi Zero 2W.

4.2 Backbone

Our backbone follows the same principle as YOLO11's: a series of strided convolutions that progressively downsample the image while increasing channel depth. Early layers capture edges and textures, while deeper layers capture the overall shape of the pad. The key difference is that we replace YOLO11's C3k2 blocks with the lighter blocks described below, reducing the parameter count.

4.3 Depthwise Separable Convolution (DSConv)

A standard convolution looks for spatial patterns and mixes all the channels together in one expensive operation. A DSConv splits this into two cheaper steps. First, a depthwise convolution looks for spatial patterns, handling each channel on its own with no mixing between them. Then a pointwise (1x1) convolution combines all channels at each pixel without any spatial work. The result is equivalent to a standard convolution at a fraction of the cost.

4.4 Cross Stage Partial Block (CSP)

The CSP block is our lightweight replacement for C3k2. It splits the input in half along the channels. One half passes through two DSConv layers for feature learning, while the other half skips them entirely. The two are then concatenated and mixed with a 1x1 convolution. The skip path preserves the original signal and gives gradients a clean route during backpropagation, similar to a ResNet.

4.5 Spatial Pooling

After the backbone, we pool the final feature map to a small 4x4 grid. We avoid collapsing to a single 1x1 value, since that would average away all spatial information and leave the model knowing whether the pad is present but not where it is. The 4x4 grid keeps a coarse sense of position while staying small enough for a lightweight head.

4.6 Head

The head is a small two-layer fully connected network that produces five outputs: a confidence score for whether the pad is present, and four normalized coordinates (x1, y1, x2, y2) for the box corners. The box coordinates pass through a sigmoid to stay between 0 and 1. The confidence stays a raw logit during training, since the loss expects logits, and passes through a sigmoid at inference to become a probability

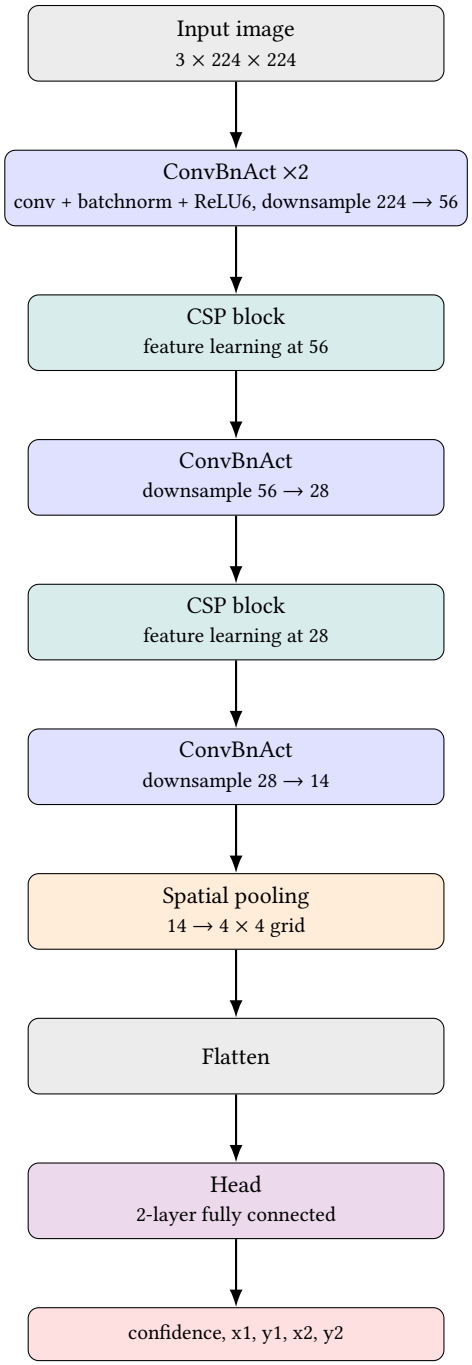

Figure 2. Custom student model architecture.

5 Training

Our teacher model was a standard YOLO11l fine-tuned for 50 epochs on a custom data set using an nvidia RTX 6000 Ada GPU. We organized our data on Roboflow, a web service which integrates models such as Meta's SAM2d to allow automatic labelling from natural language prompts. Roboflow then converted the annotations to the format used by YOLO's training architecture. We used the normal ultralytics python framework for our fine-tuning. Our distillation used KL divergence between teacher and student logits during forward passes on our fine-tuning dataset to generate training loss. We fine-tuned the model on our laptops using an NVIDIA RTX 500 Ada GPU. Distillation took approximately 45 minutes.

6 Results

Table 1. Performance comparison on test image (50 runs)

Metric	YOLO11n	Custom Model
Params	2,624,080	283,541
Size (MB)	10.010	1.082
Avg (ms)	105.07	1.14
Best (ms)	104.62	1.11
Worst (ms)	108.21	1.30
FPS	9.52	880.45

When tested against the YOLO11n model, our custom architecture took up about 10 times less RAM, commensurate with its redu ced parameter count, and was able to run inference around 100 times faster when run on our test data set; YOLO11n took 105ms on average over 50 images compared with only 11.5ms for our custom architecture. On test set images, our custom architecture was able achieve highly satisfactory localization of the landing pad, as can be seen in figures 2 and 3.

When run on the pi, our inference time was an average of around 180ms over a 1 minute test run. Due to issues with the serial connection between our pi and the flight controller, we were not able to perform a fully integrated test; we expect that under those circumstances, inference time would slow somewhat due to the cpu and ram usage of the control system. However, for detection of a static object, as is our use case, this is an acceptable frequency given the low operating speed of the drone. In order to simulate the visual field of flight, the drone was manually held over the platform (see figure 4).

Examples of inference from the pi are shown in figures 5, 6, and 7. Notably, we found the model to be somewhat less reliable when run on the drone, and to have notably worse localization performance. There are several likely explanations for this. Firstly, some degradation is to be expected given the magnitude of reduction we applied to the model. Secondly, due to delays receiving the proper ribbon cables for connecting the pi to the camera module, our data was collected using phone cameras in portrait mode. While the angle and lighting were similar, our model resizes all images to 224x224 at inference; the differing aspect ratios between phone and pi camera images would therefore result in the landing pad being vertically compressed in our training images relative to the pi camera images once scaled; this particularly accounts

Figure 4. Example of rejection in borderline case

Figure 5. Integrated evaluation setup

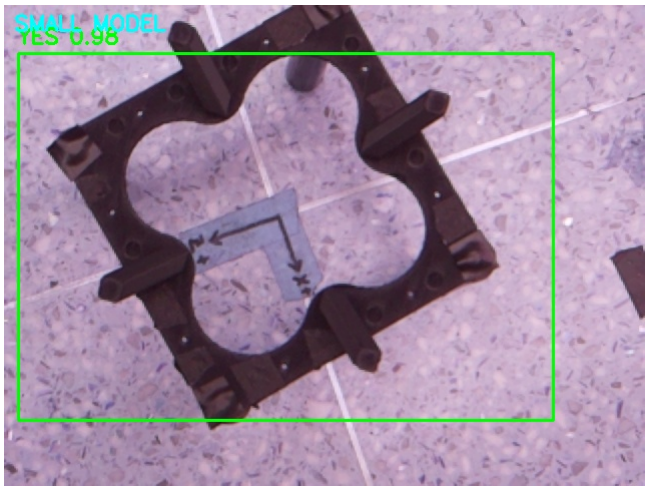

Figure 6. Example of successful detection from the pi

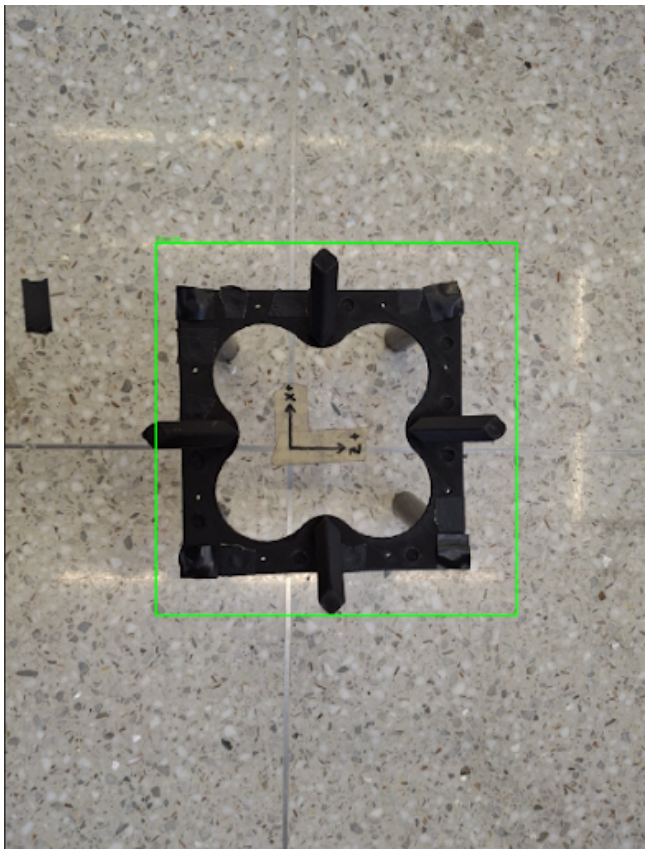

Figure 3. Example of rejection in borderline case

for the notably short and wide bounding box in figure 5.

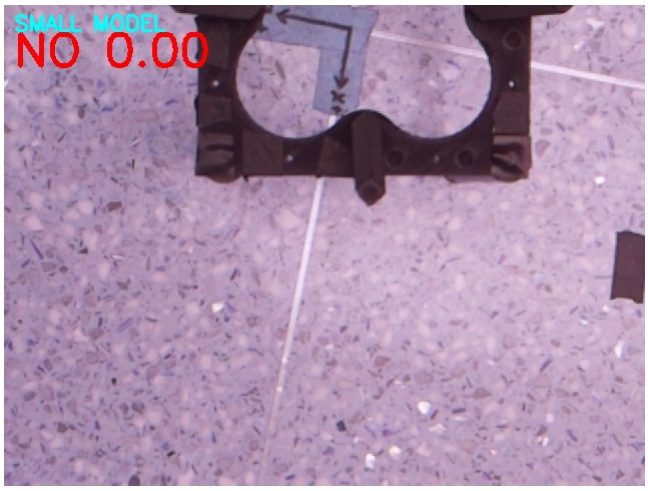

Figure 7. Example of rejection in borderline case

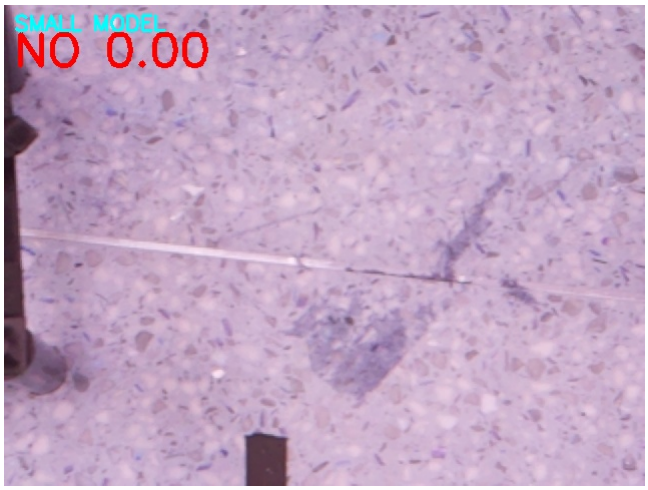

Figure 8. Example of successful non-detection from the pi

YOLO models can also be susceptible to changes in lighting condition; the drone area is underneath a skylight and our training images were taken at night, while testing happened during the day. Additionally, the pi camera module does not have an IR filter, resulting in a red tint on the images, which likely threw the YOLO model to at least some degree. Finally, our training data only included a small number of examples of the pad partially clipped out of the frame; were this model to be run on the fully functional drone, this would be a priority to remedy, as these partial detections would be important for initial localization.

7 Code Availability

The full implementation, including the custom architecture, training scripts, and distillation pipeline, is available at https://github.com/theoHC/yolo-berry-distillery.

References

https://docs.ultralytics.com/models/yolo11